Software Safety: A Definition and Some Preliminary Thoughts

by

Nancy G. Leveson

Technical Report 174

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

April 1981

This work was supported in part by a contract from Hughes Aircraft Co. (7-656146-T-DS)

ABSTRACT

Software safety is the subject of a research project in its initial stages at the University of California Irvine. This research deals with critical real-time software where the cost of an error is high, e.g. human life. In this paper software techniques having a bearing on safety are described and evaluated. Initial definitions of software safety concepts are presented along with some preliminary thoughts and research questions.

Software Safety: A Definition and Some Preliminary Thoughts

Nancy G. Leveson

Information and Computer Science
University of California, Irvine

Reliability has been the subject of research since the early days of computers. Although much progress has been made in this area, at the same time the complexity of computerized systems and applications has increased dramatically. Since errors and complexity are intimately related, the problems with which reliability techniques have had to cope have also been increasing with time, and, unfortunately, techniques to handle errors have not kept up with the increasing possibilities for introducing them.

The problem of reliability becomes even more aggravated as computer applications increasingly include areas where the consequences of failure are serious and may include loss of life and property. Examples of such applications are aircraft flight and traffic control, nuclear power plants, monitoring of critically ill patients at hospitals, weapons and defense systems, and manned space flights. With the advent of computer usage in these critical real-time applications, a new dimension was introduced into reliability -- the cost of errors. When the cost of an

error is high, for example a human life, not only do reliability factors start to take precedence over other quality factors on which software and hardware can be judged, but techniques which may have been cost prohibitive suddenly become more practical. It will be argued herein that software safety is a distinct facet of software quality and, because of its inherent severe cost factors, needs to be considered as a separate and important measure of system quality.

In this paper, the problem of software safety will be examined and a research plan outlined. First the problem of errors will be discussed and extant reliability techniques with a bearing on safety described and evaluated. Then an initial definition of software safety will be outlined and some preliminary thoughts and research directions presented.

Errors

Errors can be divided into two types -- those which arise from hardware failure and those which originate in software. Although a hardware error may result in a software error or in detection and handling through software, the definition of software errors will be limited to errors in software specification, design, and coding. The distinction can get cloudy. For example, a hardware error such as bad input data caused by a faulty input device may ultimately lead to a failure of a software routine.

Although admittedly somewhat arbitrary, this type of error will be classified for our purposes as a hardware error. A legal but unexpected input which leads to a failure will be classified as a software error. The distinction is made not on the basis of where the error should be handled (which in many cases could be either hardware or software), but rather as to where the error originated and what would have to be fixed to eliminate it. In the case of a faulty input device, the repair of the input device would be warranted. In the case of a legal but unexpected or unusual input, the software would be modified.

The problems of hardware errors and hardware fault-tolerance has been widely studied [for a good summary see AVI75], and computer systems which tolerate hardware component failure can be constructed by introducing redundancy. However, less progress has been made toward the problem of dealing with design errors. Until recently, the design of hardware has been relatively simple, and it has been possible to detect errors through comprehensive testing. With the advent of VLSI, however, this fortunate situation may soon be changed.

Most of the complexity in systems has been incorporated in the software where comprehensive testing is not possible for large systems. Thus most of the reliability problems in computing systems can be traced to software design errors[20]. This research will focus primarily on software

errors. In practice, of course, the goal is a reliable computing system which is free from or can tolerate all errors -- hardware and software. A combination of the techniques for hardware and software reliability may help meet this goal.

Software errors have been regarded as a temporary problem. It seems to be widely felt that as soon as adequate methodologies for program development and validation can be devised, software errors will no longer be a problem. But progress in developing these methodologies has been slow, and error-free software may not be a realistic goal. Since human programmers are not infallible, it is unlikely that we will be able to construct large error-free systems until automatic program synthesis becomes a reality. Even automatic programming may not be the panacea we seek. Errors are made in all stages of program production. There is evidence that more errors occur during the requirements and design stage (over 60%) than during coding (less than 20%) [LIP79].

Errors caused by human mistakes can be introduced at any stage of the software life-cycle. The requirements specification may be incomplete or ambiguous; the system design may not adequately implement the requirements either through misinterpretation or errors in specifying interactions among the parts of the system; programmers may misinterpret the design specification; and so on.

Furthermore, critical real-time software may have additional requirements not found in other systems which add to the complexity of the resulting software (and hence the potential for error). For example, there may be a time limit on the execution of the program, i.e. output may be required within a few milliseconds after the input stimuli. This precludes the use of many techniques for error handling which are used in situations where accuracy is just as important, e.g. accounting systems, but where time is not so severly constrained.

Software reliability techniques may be divided into four categories: 1) techniques to prevent or eliminate errors while the system is under development, 2) techniques to test and validate software after it is written but before the system goes into production, 3) techniques to model and measure reliability, and 4) techniques to cope with residual design flaws, coding bugs, hardware errors, and user errors after the system has been validated. The first two techniques involve primarily program development, the third measurement, and the fourth what is known as software fault tolerance.

Program Development

Although development techniques are often separated from validation, there is a danger is doing so. Parnas [PAR77] has said that no methodology is useful unless it

includes a means for recording intermediate design decisions and for verifying the correctness of each of these decisions. The usual diagram of the software lifecycle is misleading and unfortunate in including testing and validation in a step after coding. Since it is widely recognized that the earlier errors are caught, the easier they are to fix [LIP79], a test and verification step should be included in the diagram after each step of the cycle. In order to emphasize this point, in this paper program development techniques will be separated into those which attempt to prevent the inclusion of errors and those which attempt to locate and eliminate the errors which have already been included.

Approaches which attempt to encourage error-free construction of software systems include top-down design and other design techniques and tools, structured programming, formal specification methods and tools, programming standards, and project organization and management techniques. Although there is great concensus that these techniques do produce "better" software, they must be used by humans and humans are fallible. There are even the designers and programmers that Parnas has called the "methodology invariant" for whom no methodology will be helpful. Therefore, software will contain errors for some time to come.

Validation methods attempt to find errors after they have been included in the system and, hopefully, to eliminate them. These methods include the use of program debugging and debugging tools, verification of formal specifications, design reviews and walkthroughs, testing methods and automated testing tools, and techniques for proving program correctness.

Validation methods present some serious problems in assuring reliable software which will probably not be relieved by devising better procedures. First, for large systems, exhaustive testing of all possible cases is impossible. Furthermore, the process of designing test data is difficult and is itself subject to error. Also, there is no way to determine when testing or review is complete. Reliability estimates are used for this purpose, but at present are far from reliable themselves.

Formal proof techniques are an attempt to remedy some of the limitations of testing. Even assuming that mechanical verification systems were available and usable and that cost was unimportant, a formal proof only demonstrates the correctness of the program with respect to the specifications. There is no guarantee that the specifications are correct, and, in fact, writing formal specifications is a difficult and error-prone process [GER76]. Furthermore, the complexity of proving a large system correct is so great that the process must itself be

error-prone.

But the greatest limitation of validation techniques is that they cannot guard against imperfect execution environments. A program can only be validated for given environmental assumptions. Software errors may be caused by undetected hardware errors such as transient faults causing mutilation of data, security violations, human mistakes during operation and maintenance, errors in underlying or supporting software systems, or interfacing problems with other systems such as timing errors. Just as all correct paths through a program cannot be tested for a complex system, neither can all environmental conditions.

Reliability Modelling and Measurement

Most of the techniques for modelling or measuring the reliability of software have been adapted from techniques originally devised for hardware modelling. The basic approach is to rate the reliability of a software system by estimating the number of errors in the system [SW78]. The applicability of these hardware techniques to software has been questioned, and this issue will be covered in a later paper.

Other techniques specific to software for measuring reliability have been proposed. The basic assumption is that reliability is inversely proportional to program complexity. Three types of complexity measurements have

been proposed [RAU]: 1) textual complexity based on a static analysis of program text (e.g. the number of operators and operands), 2) structural complexity based on static analysis of program graphs (e.g. a count of decision instructions in the program), and 3) structual complexity based on execution behavior (e.g. the number of program paths). The applicability of these measures to reliability is unproven.

Software Fault Tolerance

The methods described above have the common goal of eliminating errors prior to the operational use of the software system. As argued above, however, removal of all errors and perfect execution in imperfect environments cannot at this point in time, and probably never will, be guaranteed using these methods. For this reason, software fault tolerance is a necessity in real-time critical systems. The software fault tolerance approach assumes that errors will occur and attempts to increase reliability by ensuring that the software will continue to function successfully in spite of the presence of errors.

Most sophisticated software systems include facilities for dealing with errors that are detected at run-time, e.g. input editing, auditing, and logging facilities in data processing applications, integrity checking and recovery in data base management systems, and exception-handling and

checkpointing facilities in operating systems. In environments requiring continuous service, e.g. operating systems, telephone switching systems, and space flight systems, sophisticated ad hoc facilities have provided a large measure of protection from hardware failures as well as from some software errors. These techniques are, however, ad hoc, and the aim of research in software fault tolerance has been to propose and evaluate methodological approaches.

In general, run-time error facilities have three aspects: 1) error detection, 2) limitation of damage caused by the failure, and 3) recovery.

Error detection has been widely studied and practiced in both software and hardware. Most hardware techniques involve redundancy -- component redundancy such as voting systems or redundant information such as parity or Hamming codes. Detection facilities in software also use "component" or activity redundancy and redundant information. Activity redundancy may take the form of replicating some activity to check the results or perhaps reversal of the activity to calculate what the input should have been. An example of redundant information might be some extra information kept about data structures. Software detection facilities may also include other types of checks such as as reasonableness checks and interface checks (i.e. checking the interactions across interfaces).

Damage confinement has been studied extensively with respect to operating systems. Techniques depend upon constraints on information flow in the system and are known as protection mechanisms. Once damage has occurred, in order to recover there must be some way to undo the side effects of erroneous operations.

Information redundancy is used to reconstruct damaged data and may be in the form of backup copies or redundancy in the representation of the data. Taylor, Morgan, and Black [TMB80] have investigated the use of redundancy in data structures to detect and recover from errors. There are three drawbacks to this approach. First, redundant information increases the complexity of the data structure and of update routines and, in turn, the increased complexity may itself lead to errors. Also, correct update routines may propagate erroneous changes. Taylor et.al. have shown that in many highly redundant structures, the rate of error propagation is directly proportional to the detectability. Finally, erroneous changes may be made to multiple fields which may limit the ability to detect error through inconsistency.

An alternative to structural redundancy is to keep backup copies. Either complete checkpoints may be taken frequently or, alternatively, backup copies made infrequently and only changes recorded between checkpoints. Randell's "recovery cache" mechanism [RAN75] involves making

backup copies at the entry to each block. Sophisticated techniques for checkpointing are used in data base and data processing systems. The drawbacks to backup redundancy include cost, the time needed to reprocess transactions against the data, and the possibility of propagation of errors. However, backup redundancy is probably more practical than structural redundancy because of the simplicity of implementation and the reliability of recovery without error. Not only can errors be introduced in the added software needed to process the redundant information in the data structure, but there is no guarantee that errors can be undone using structural redundancy.

Once an error is detected, measures must be taken to deal with it. Software recovery techniques have usually involved exception handling facilities. Every complex system can be viewed as being composed of multiple levels of abstraction. The primary goal of exception handling is to make the failure transparent to higher levels. Barring this, the system should be put in a consistent state so that recovery is possible. The failure is then reported to a higher level which can initiate "intelligent" error-handling action and, hopefully, make the failure transparent to even higher levels (including the user of the system).

Exception handling has three purposes [GOO75]. The first is to deal with an operation's impending or actual failure. Goodenough has noted that there are two types of

failures -- range failure and domain failure. A range failure occurs when an operation is unable to satisfy its output assertion, i.e. its criterion for determining whether it has produced a valid result. On the other hand, a domain failure occurs when an operation's inputs fail to pass certain acceptance tests, i.e. the input assertion. A second use for exception handling is to indicate the significance of a valid result or the circumstances under which the result was obtained. Exception handling may also be used to allow the invoker to monitor an operation, e.g. to measure computational progress or to provide additional information and guidance in the event that certain conditions should arise. Exception handling procedures have been described extensively in the literature [see for example GOO75, WAS78, LS79].

Some hardware experts have argued against using exception-handling to recover from errors and instead have proposed applying standard hardware techniques to software [RAN79, MSR75, AVI75]. The result has been called "software fault tolerance."

All hardware fault tolerance is based on the use of redundancy, both for error detection and recovery. This redundancy may be spacial (concurrent execution of multiple copies) or temporal (consecutive execution of the same component). Hardware errors may be classified as either permanent or transient. Permanent errors are usually due to

some kind of "aging" process, and spatial redundancy is sufficient to recover. Diagnosis of the error will not help except in ultimately repairing the component. Transient errors, also caused by aging or by some external event (e.g. power surges) may be handled with either spatial or temporal redundancy.

Proponents of the software fault tolerance approach such as Randell [RAN75, RAN79] have argued that hardware techniques should be directly adapted to software and that no attempts should be made to diagnose the particular fault that caused an error or to assess the extent of any other damage the fault may have caused, i.e. exception handling is inappropriate for handling design errors. Recovery actions need merely to return the system to a prior state (hopefully one which precedes the error), and then go forward again with an alternate piece of code.

But one should not be too hasty in accepting this conclusion. Many large software systems are much more complex than hardware systems, and the underlying assumptions about causes of errors are different in software and hardware. Software errors are usually errors of design. More sophisticated types of facilities which take advantage of the algorithmic power of software may be necessary to detect and recover from software design errors than the relatively simpler errors which occur in hardware. Much more justification and study is needed before it is possible

to say with certainty that hardware techniques are appropriate for software.

Two approaches based on hardware fault tolerance have been proposed -- n-version programming and the recovery block.

In n-version programming, multiple versions of an algorithm are executed simultaneously and the results compared. If the results differ, voting or other strategies can be used to select one result [ELM72, CA78]. In hardware, since failure is due to fatigue, modular redundancy with voting circuits can greatly increase reliability. In software, however, multiple versions of algorithms that contain design errors does not increase reliability. The success of this approach then is dependent upon the complete independence of the n versions. This implies the use of different algorithms, programming languages, and support systems. Since a large percentage of errors can be traced back to incomplete, inconsistent, or ambiguous requirements specifications, independent development must start at the requirements specification stage. A further requirement for an n-version system is m computers (where m is greater than or equal to n) that are hardware independent yet able to communicate efficiently for rapid comparison of results.

Dual (2-version) programming has been tried in two European projects. Swedish State Railways has a point switching, signal control, and traffic control system in the Gothenburg area which uses redundant programmning [TAY81]. If two programs show different results, signal lights are switched to red. A nuclear reactor project was jointly developed in Finland and Norway using dual programming [RAM]. There were two implementation approaches and two programming languages in two different countries. A common specification was used, however, and two different interpretations of the specification resulted although the discrepancies were found prior to implementation. European experience seems to indicate common errors in about half of the redundant software systems developed to date [TAY81]

A second method of providing software tolerance uses recovery blocks [RAN75]. A recovery block consists of a regular programming language block (called the primary block), an acceptance test, and a sequence of alternate blocks.

A: ensure <acceptance test>
by <primary block>
else by <alternate block>
else by ...

The acceptance test is a logical expression which is evaluated to determine if the result of a block is correct.

If a primary or alternate block does not complete (because of an error or expiration of time limit) or fails the acceptance test, the state is restored to just prior to entering the recovery block, and the next alternate (if there is one) is entered. If all alternates fail to pass the acceptance test, recovery is attempted at the level of the next enclosing recovery block. If the acceptance test is passed, control passes to the statement after the recovery block. Prior states to be used for rollback are stored in a "recovery cache."

Although experience with recovery blocks is limited, alternate modules have been used, in an ad hoc manner, in space applications [HEC76].

An advantage of the recovery block is its simple control structure. Also, the alternates need not be identical to the primary block. In some cases, there may be alternate algorithms for achieving the same ends, perhaps some more "desirable" than others. There may also be a fast heuristic which usually works which can be backed up by slower but always correct algorithms. Finally, in systems undergoing maintenance, further errors are sometimes introduced in the maintenance process. Sometimes it is possible to keep an older (and perhaps more robust) version as a backup routine.

Randell has described some of the problems of the recovery block technique. Restoring the state before switching to an alternate may be difficult to implement efficiently and may require hardware aids. Also, structuring systems of communicating processes adds complications. Restoring the state of one process may require that other processes with which it has communicated restore their states, thus creating a "domino effect."

There are further drawbacks in applying recovery blocks to real-time critical systems. Time requirements may be such that switching to an alternate routine and starting again means that results will arrive too late to be useful. And in some situations backup to a prior state may be impossible. This is the "please ignore approaching torpedo" problem.

Table I summarizes the important differences between n-version programming and recovery blocks.

	n-version	recovery block
calculations	same	different possible
execution	parallel	successive
error-detection	voting schemes	acceptance test

Table I

The relative advantages of exception handling vs. hardware-oriented approaches for software fault tolerance is an open question. Exception handling research has been the more general of the two in that it has included attempts to predict and deal with anticipated or likely faults along with fault diagnosis and damage assessment. The software fault tolerance approach, on the other hand, has borrowed hardware fault tolerance techniques where diagnosis and damage assessment is not important for recovery, and anticipated and unanticipated errors are dealt with in the same fashion.

Software Safety

Now that reliability and reliability techniques have been described, it is possible to define software safety and to differentiate it from reliability. There does not seem to be any formal definition of software safety in the literature so the first step in the project will be to attempt a definition.

To define safety, it is necessary to first define some common terms in reliability such as "failure," "error," and "fault." Generally, a failure is said to have occurred when the behavior of a system deviates from its specifications. Reliability is a measure of the success with which a system comforms to some specification of its behavior. Although the terms error and fault are used differently in the

hardware and software literature, the IEEE terminology task group has suggested that an error be defined as a causative agent and a fault as a symptom or manisfestation of an error [LIP79]. While in hardware errors are caused by fatigue and may appear in previously error-free components, all software errors can be traced back to human error and are present from the inception of the software object (except for errors added during maintenance). However, the concept of "chain of effects" is useful in relation to software errors since one error may lead to another error, e.g. an error in an algorithm may lead to an error in a state.

One is first tempted to define safety as reliability. It is true that a more reliable system is usually safer than a less reliable system. Obviously they are closely related, but differences do exist. First, the goals of the two are different. The goal of reliability is lack of unintended function. The goal of safety is lack of unsafe operation. A system may perform in an unintended manner but still not result in unsafe operation.

A second difference is that reliability is concerned with failures. Safety is concerned with the consequence of failures in terms of economic or human cost. Some failures are more serious than others, and, in turn, this implies that some errors are more serious than others. Reliability theory currently tries to quantify errors, safety implies that errors must be qualified. The necessity for some kind

of penalty cost analysis in reliability has been noted by Littlewood [LIT80] and Cheung [CHE80]. But so far, reliability models simply count failures which, in effect, gives the same cost to each.

Finally, it is important to separate reliability and safety because, in some cases, these may actually be conflicting goals, and proper decision making will require knowledge of this fact. Safety procedures in a weapons system may involve disabling the detonation facilities of the weapon. In the absurd case, disabling or shutdown could always occur after firing. This would make the weapon system 100% safe but 0% reliable. In other more realistic circumstances, an error may be detected and a decision may need to be made as to whether to switch to backup safety procedures or to try to fix the error. Attempting to fix the error would result in increased probability of intended function but perhaps also might result in decreased safety (e.g. if recovery is attempted, there may no longer be a guarantee that enough time will be left to successfully implement safety procedures). Decision making here will involve questions of morality, politics, etc. which must be examined by those equipped to do so. But the important point is that less information is available to make these decisions if safety and reliability are lumped together in one measure.

A safety failure (commonly termed a mishap in system safety) will be defined as a failure which leads to casualties or serious consequences. The definition of serious consequences will have to be left up to the system designer. It obviously includes injury or death. It may also include the destruction of property or any other undesirable consequence the avoidance of which the designer of the system considers as or more important than the correct operation of the system. In some cases, such as software for manned space craft, nearly every failure may be a safety failure. An example of a non-critical failure might be a failure in an archival routine intended only to provide data for post-mission analysis.

An unsafe state will be defined as one for which there are circumstances where further processing will lead to a safety failure which we do not attribute to a subsequent error. A safe system will be defined as one which prevents unsafe states from producing safety failures.

Reliable software has had two distinct meanings [YEH76]. Most often, software reliability is equated with correctness. A system is correct if it meets its intended function. This implies that the specifications are satisfied, e.g. the input and the environment in which the software runs meet the requirements of the specifications. Attempts to make software fault-tolerant have involved the quality of robustness. Software is robust if it can

continue to give adequate service even when faced with unspecified or unexpected circumstances (e.g. hardware failure, bad data, or software error). Another quality with which software can be judged, safety, is being proposed. It is distinct from both correctness and robustness although obviously related to both. One goal of this research is to clarify this relationship. The difference between correctness (the standard meaning for reliability) and safety has been discussed. It remains to try to differentiate between fault-tolerance and safety.

In the hardware field, a fault-tolerant computing system is one which carries out its program correctly in the presence of operational faults. Thus the aim of fault-tolerance is to continue to provide full performance and functional capabilities. The goal of fail-soft procedures (also called partial fault tolerance or graceful degradation) is to continue operation but to provide only degraded performance or reduced functional capabilities until the fault is removed. The goal of fail-safe procedures is to limit the amount of damage caused by a fault. No attempt is made to satisfy the functional specifications except where this is necessary to ensure safety.

In some critical real-time software, e.g. that in the space industry, a fault-tolerant position is taken -- a system either works or it does not, and failure of a

critical system is intolerable. In other situations, e.g. a patient monitoring system, fail-soft procedures may suffice. In weapons systems, fail-safe procedures may be the most realistic. Most systems take only one view. But in some large, complex systems, these procedures might be used to interface with each other (fail-safe or fail-soft procedures to back up fault-tolerant procedures in event of their failure). Also, one of these procedures might be appropriate for one part of the system while a different procedure might be appropriate in another part.

In summary the following working definitions will be used:

failure - when a system fails to deliver "expected behavior."

safety failure - a failure which lead to casualties or serious consequences. A serious consequence is any undesirable event which the designer considers to be as or more important than the correct (reliable) operation of the system.

safety - a measure of the ability of the system to avoid safety failures.

unsafe state - one for which there are circumstances where further processing will lead to a safety failure.

safe system - one which prevents unsafe states from causing safety failures.

fail-safe system - a system which limits the amount

of damage caused by a fault. No attempt is made to satisfy the functional specifications except where necessary to ensure safety.

Preliminary Thoughts and Questions

Starting with the definition of software safety and related terms given above, the goals of the project are to examine the following questions and preliminary ideas.

1) Design Considerations

-- Can safety techniques be classified? For example, sometimes shutdown of components which might cause harm (reconfiguration) is appropriate while in other situations something must be done to avoid harm (ditch rocket in ocean or blow it up in midair).

-- Is it possible to design so that critical and non-critical functions are separate and the system is reconfigurable? Some errors in non-critical functions will not affect overall safety, e.g. bad computation, but some will (e.g. infinite loop, hardware failure) and perhaps it is necessary to terminate execution of non-critical sections. A "supervisor" might be used to control execution so that unsafe but non-critical modules can be avoided.

-- Critical modules from a reliability viewpoint may not be those from a safety viewpoint. The user-oriented reliability model [CHE80] indicates that the program modules used most often during execution time probably are the critical modules from a reliability point of view. Is it possible to identify critical modules from a safety point of view? If so, how?

-- Where should fault-tolerance be used and where fail-soft or fail-safe procedures? Because of the high cost of complete software fault-tolerance, it will be desirable to protect non-critical modules only to the extent that faults in their execution do not impact the critical functions.

--Is it possible to provide a "safety kernel?" The kernel approach has been used successfully in operating systems, e.g. a security kernel. The kernel might be used to organize recovery in other segments and/or might contain critical functions.

-- Is there one methodology and structure to handle both anticipated and unanticipated errors and to handle fault-tolerance, fail-soft, and fail-safe?

--Critical modules should be independent of other modules, the structures simple and elegant, and efficiency sacrificed if necessary. Are there critical and non-critical errors within critical modules?

--How will fail-safe measures be made fail-safe? Perhaps by extra testing or proof of correctness? Does the addition of fault-tolerance, fail-soft, or fail-safe procedures add to the complexity of the system and thus reduce its reliability and safety? Who checks the error checking? What happens when an error occurs in an error checking mechanism?

-- Is there some design or system structuring technique such as state transition diagrams denoting illegal transitions which would be helpful?

-- Upon detection of an error, is it possible to place the system in a consistent state so that no damage will be done independent of whether the error is repaired or not and then try to repair it instead of vice versa?

--What facilities can be put into the program and what into the interpreter or underlying system? In other words, what is the appropriate level(s) for handling safety? Usually one tries to design so that error detection and handling at one level is invisible at the earliest possible higher level. Does this apply with safety?

--Are the protection mechanisms of operating systems (or something like them) applicable for preventing a failure in non-critical functions from interfering with critical functions?

-- Are there architectures, e.g. data flow, which might be superior for safety? What about functional programming languages where there are no side effects to undo for backup? What extensions to programming languages (e.g. Ada) are necessary to handle safety requirements?

--Is it possible or necessary to locate the original error so that it can be avoided? The detection of an erroneous state does not necessarily identify the cause of the error.

--Damage assessment is the procedure which determines the extent of damage in the case of an error. How can one tell if a state is unsafe or merely erroneous? Must this be based on a priori reasoning? Or can the system itself help by means of exploratory procedures?

2) Development considerations

--What development procedures will lead to systems which are safe? It appears that it is necessary to work toward safety during the entire life cycle.

--The requirements specification should include safety requirements and an analysis of possible types of errors. Is there anything else? How should this be done?

-- Perhaps a new attitude is necessary in design and coding, that is, to assume errors will exist and to consider while designing and programming how to recover from these

errors. A proof-guided methodology might be devised for designing safety procedures. Can safety requirements be traced through coding?

-- Are n-version programming or recovery blocks appropriate for safety? In n-version programming, how can independence be ensured? Perhaps it is not necessary to replicate the entire system. In SIFT [CAR79], most critical functions are assigned to several modules while less critical tasks are placed in a lesser number of modules.

-- In testing and validation, how much of the available resources should be put into critical functions and safety techniques? How much into correctness? Robustness? Safety features should be thoroughly tested. Diagnostic tests which deliberately violate requirement or design specifications are important. Is it possible to specify procedures for accomplishing this? Are proof of correctness, exhautive testing, or stricter acceptance tests appropriate for safety critical modules?

-- In maintenance it may be possible to indicate modifications which may have a traumatic effect on the safety of the system and to avoid as much as possible modifying safety critical modules when considering maintenance alternatives (although this should not preclude modifications which will improve safety).

3)Applicability of hardware techniques

-- The basic assumptions in hardware reliability include: a component is correct before a fault; most errors are caused by fatigue, not design; individual copies of components are independent; it is much more important to be able to recover from errors than to prevent them. What relationship do these hardware assumptions have to software? Can error detection techniques and rollback provisions be adapted to software failures?

-- Redundancy guarantees high reliability of hardware components because errors are caused by fatigue, not design. Does redundancy guarantee such a high reliability with design errors? In fact, what does redundancy mean in software terms? Is it the only or best method for reliability and/or safety in software?

4) Modelling and Measurement

-- Are hardware reliability measures applicable to software? What relevance do they have to safety?

-- Many of the software reliability models attempt to measure the number of residual errors. Is this reasonable for reliability? For safety?

-- What about other techniques which might be more relevant to software?

-- Is it possible to simply apply regular reliability models to a subset of errors, ie. safety critical errors?

-- How should fault-tolerance be modelled and measured?

-- Can a practical reliability model be formulated which covers both software errors and hardware faults? A safety model?

5) Types of Errors and Error Detection

-- There are three types of system errors: hardware errors detected and handled by hardware, software-handled hardware errors, and software errors. Is this a reasonable classification? What other types of error classification are relevant to software safety? Is error classification dependent upon the perspective and structure imposed on the system?

-- What kinds of errors occur in software? In specifications? How can safety critical errors be identified?

-- Safety seems to be intimately connected to (perhaps dependent on) error detection. What are the characteristics of good error detection facilities? What kinds of error detection techniques are needed for detecting errors in design? Coding errors? Bad input data? Hardware errors? Where should error checking be done, e.g. in application programs, in special support software, in the operating

system, in the hardware? Are different types of techniques more applicable to safety errors than to other errors? What kinds of language facilities are useful for error checking? What if there are errors in the error-detection facilities? Who checks the checks?

-- It is possible to define safety assertions (or checks) as assertions on critical code. Is this useful?

-- Is it necessary for the program to understand the semantics of the desired action in order to check for errors? Are any of the formal techniques for specifying programming language semantics applicable? What is the relationship between safety checks and assertions used in correctness proofs?

-- Can some classes of assertions and checks be removed through appropriate testing? What types cannot ever be removed?

Conclusions

A preliminary definition of software safety and related terms has been attempted and some questions related to this area presented. Obviously the answers to all these questions cannot be determined in the remaining months of this project. Only a beginning can be made. Focus will be placed on the more basic questions which need to be answered first and an attempt made to formulate a model with which to

view the basic concepts.

References

[ALS79] Anderson, T., P.A. Lee, and S.K. Shrivastava. System Fault Tolerance, in T. Anderson and B. Randell (eds.) Computing Systems Reliability, Cambridge University Press, 1979.

[AVI75] Avizienas, A. "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," Proceedings of the Int. Conference on Reliable Software, SIGPLAN Notices, vol. 10, no. 6, 1975.

[CAR79] Carter, W.C. "Hardware Fault Tolerance," in T. Anderson and B. Randell (eds.) Computing Systems Reliability, Cambridge University Press, 1979.

[CA78] Chen, L. and A. Avizienis. "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Eighth Int. Conf. on Fault-Tolerant Computing, Toulouse, France, June 1978.

[CHE80] Cheung, R. C. "A User-Oriented Software Reliability Model," IEEE Trans. on Software Engineering, vol. SE-6, no. 2, March 1980.

[ELM72] Elmendorf, W.R. "Fault-Tolerant Programming," Digest of 1972 Int. Symp. on Fault-Tolerant Computing, pp. 79-83.

[GH75] Gannon, J.D. and J.J. Horning. "Language Design for Programming Reliability," IEEE Trans. on Software Engineering, vol. SE-1, no. 2, June 1975, pp. 179-191.

[GS78] Gannon, T.F. and S.D. Shapiro. "An Optimal Approach to Fault Tolerant Systems Design," IEEE Trans. on Software Engineering, vol. SE-4, no. 5, Sept. 1978, pp. 390-409.

[GY76] Gerhart, S.L. and L. Yelowitz. "Observations on the Fallibility in Applications of Modern Programming Methodologies," IEEE Trans. on Software Engineering, vol. SE-2, 1976, pp. 195-207.

[GOO75] Goodenough, J.B. "Exception Handling: Issues and a Proposed Notation," CACM, vol. 18, no. 12, Dec. 1975, pp. 683-696.

[HEC76] Hecht, H. "Fault-Tolerant Software for Real-

Time Applications," Computing Surveys, vol. 8, no. 4m Dec. 1976.

[HEC79] Hecht, H. "Fault-Tolerant Software," IEEE Trans. on Reliability, vol. R-28, no. 3, Aug. 1979.

[KY75] Kane, J.R. and S.S. Yau. "Concurrent Software Fault Detection," IEEE Trans. on Software Engineering, vol. SE-1, no. 1, March 1975.

[LIP79] Lipow, M. "Prediction of Software Errors," Journal of Systems and Software, 1, 1979 pp. 71-75.

[LS79] Liskov, B.H. and A. Snyder. "Exception Handling in Clu," IEEE Trans. on Software Engineering, vol. SE-6, no. 6, Nov. 1979, pp. 546-558.

[LIT80] Littlewood, B. "Theories of Software Reliability: How Good Are They and How Can They be Improved," IEEE Trans. on Software Engineering, vol. SE-6, no. 5, Sept. 1980.

[MSB75] Melliar-Smith, P.M. and B. Randell. "Software Reliability: The Role of Programmed Exception Handling," Proc. Int. Conf. on Reliable Software, SIGPLAN Notices, vol. 10, no. 6, June 1975.

[MUS75] Musa, J.D. "A Theory of Software Reliability," IEEE Trans. on Software Engineering, vol. SE-1, no. 3, Sept. 1975.

[PAR77] Parnas, D.L. "The Use of Precise Specifications in the Development of Software," IFIP-77, North Holland, 1977.

[Ram] Ramamoorthy, C.V., F.B. Bastani, J.M. Favaro, Y.R. Mok, C.W. Nam, and K. Suzuki. "A Systematic Approach to the Development and Validation of Critical Software for Nuclear Power Plants."

[RAN75] Randell, B. "System Structure for Software Fault Tolerance," Proc. Int. Conf. on Reliable Software, SIGPLAN Notices, vol. 10, no. 6, June 1975.

[RAN79] Randell, B. "System Reliability and Structuring," in T. Anderson and B. Randell (eds.), Computing Systems Reliability, Cambridge University Press, 1979

[RAU] Rault, J.C. "An Approach Towards Reliable Software,"

[SW78] Schick, G.J. and R.W. Wolverton. "An Analysis of

Competing Software Reliability Models," IEEE Trans. on Software Engineering," vol. SE-4, no. 2, Mar. 1978.

[TAY81] Taylor, B. "Letter from the editor," Software Engineering Notes," vol. 6, no. 1, Jan. 1981.

[TMB80] Taylor, D.J., D.E. Morgan, and J.P. Black. "Redundancy in Data Structures: Improving Software Fault Tolerance," IEEE Trans. on Software Engineering, vol. SE-6, no. 6, Nov. 1980.

[WAS78] Wasserman, A.I. "Procedure-Oriented Exception Handling," Medical Information Science, UCSF, 1978.

[WUL75] Wulf, W.A. "Reliable Hardware/Software Architecture," Trans. on Software Engineering, vol. SE-1, no. 2, June 1975, pp. 233-240.

[YC75] Yau, S.S. and R.C. Cheung. "Design of Self-Checking Software," Proc. Int. Conf. on Reliable Software, SIGPLAN Notices, vol. 10, no. 6, June 1975.

[YEH76] Yeh, R.T. "Guest Editorial," ACM Computing Surveys, vol. 8, Dec. 1976, pp. 355-357.